Everything you need to know to start with C

while(98) printf(“#cisfun\n”);

What is C?

C is an imperative (procedural) language.

All the code has to be inside a function.

C files are source code of your program.

You have to compile C files with a compiler (for instance gcc) to create an executable file.

Comments

#cisfun

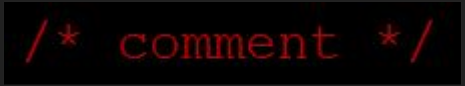
Comments

Begins with /* and ends with */

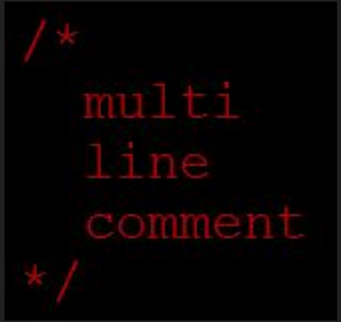
Can be inserted anywhere a white-space character is allowed

Comments don’t nest

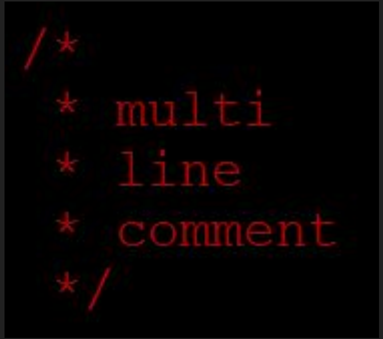
Use comments to document your code



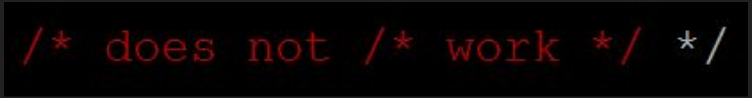
```
/* comment */
```



```
/*
  multi
  line
  comment
*/
```



```
/*
 * multi
 * line
 * comment
 */
```



```
/* does not /* work */ */
```

Variables

#cisfun

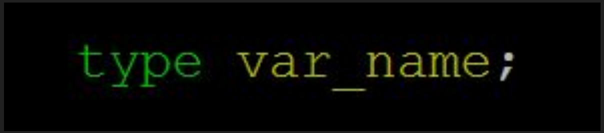
Data types | Integer types (on most 64bits computers)

Type	Storage size	Value range
char	1 byte	-128 to 127
unsigned char	1 byte	0 to 255
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
int	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned int	4 bytes	0 to 4,294,967,295
long	8 bytes	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long	8 bytes	0 to 18,446,744,073,709,551,615

Declaration

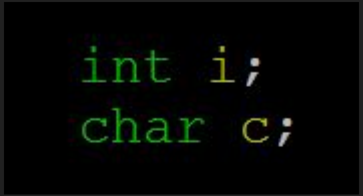
Variables always have a type.

Syntax:



```
type var_name;
```

Example:



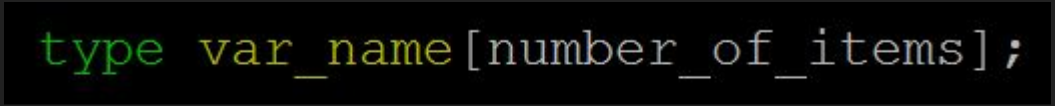
```
int i;
char c;
```

Names of variables: [a-zA-z_][a-zA-Z_0-9]*

Arrays

A succession of items of the same type in memory

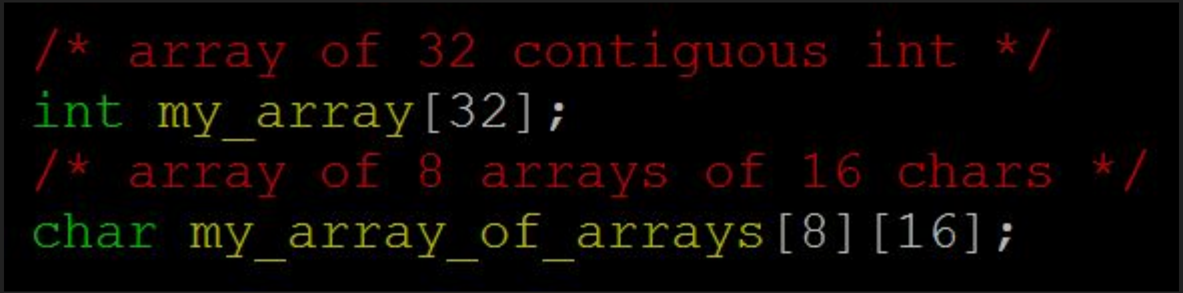
Declaration:



```c
type var_name[number_of_items];
```

Where number_of_items is a constant number (not a variable)

Example:

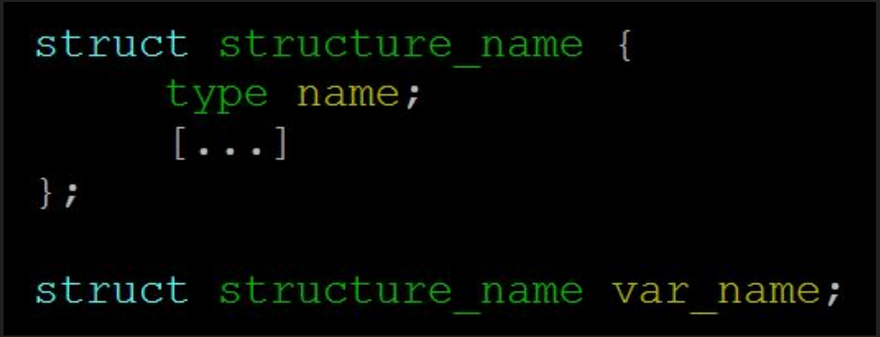


```c
/* array of 32 contiguous int */
int my_array[32];
/* array of 8 arrays of 16 chars */
char my_array_of_arrays[8][16];
```

Structures

A complex data type declaration that defines a physically grouped list of variables to be placed under one name in a block of memory

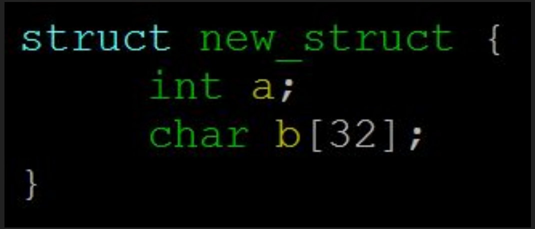
Creation and declaration



```
struct structure_name {
        type name;
        [...]
};

struct structure_name var_name;
```

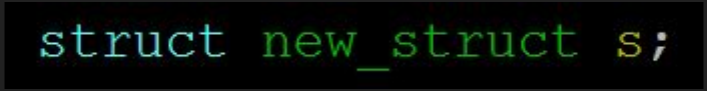
Example

Creation of a new type called new_struct



```
struct new_struct {
        int a;
        char b[32];
}
```

Declaring a variable of type new_struct



```
struct new_struct s;
```

Arrays and structures

Arrays of structures

```
struct new_struct var_array[32];
```

Structures with structures elements

```
struct student {
      char first_name[32];
      char last_name[32];
      struct address addr;
}
```

Functions, Programs

#cisfun

Functions

A program is a collection of functions

Functions:

- Sequence of program instructions that perform a specific task, packaged as a unit
- May take arguments as input information
- Compute something
- May return a result
- Functions can call functions

Parameters and return values must have a type

Syntax

```
return_type function_name(type param, type2 param2 [...])
[block]
```

Example

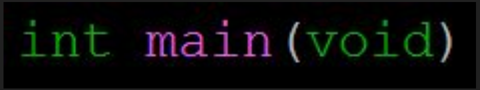
```
int func(int a, char b)
[block]
```

The entry point | main

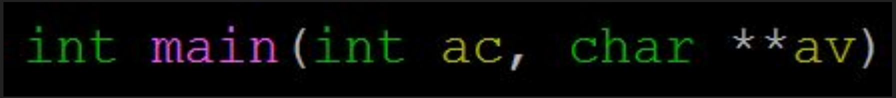
Program starts with the entry point. In C it's the main function

From the main function you can call other functions

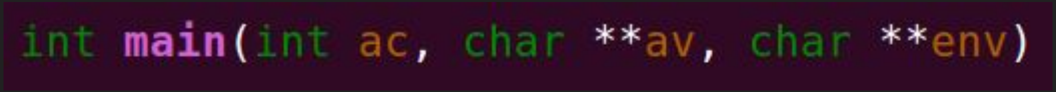
When main returns, the program stops



```c
int main(void)
```



```c
int main(int ac, char **av)
```



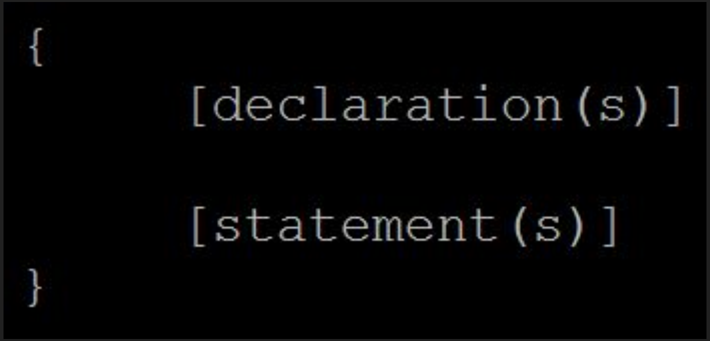
```c
int main(int ac, char **av, char **env)
```

Blocks

Blocks or code blocks are sections of code which are grouped together. Blocks consist of one or more declarations and statements.

Blocks are delimited by curly braces { and }

Syntax



```
{
        [declaration(s)]

        [statement(s)]
}
```

Blocks

Declarations

Declare variables

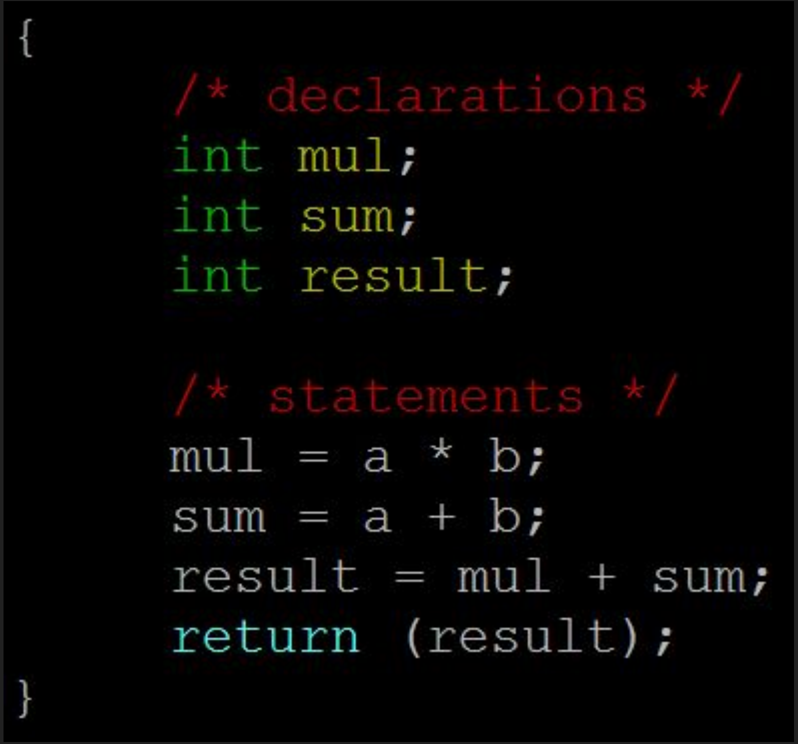
Statements

Executed in order

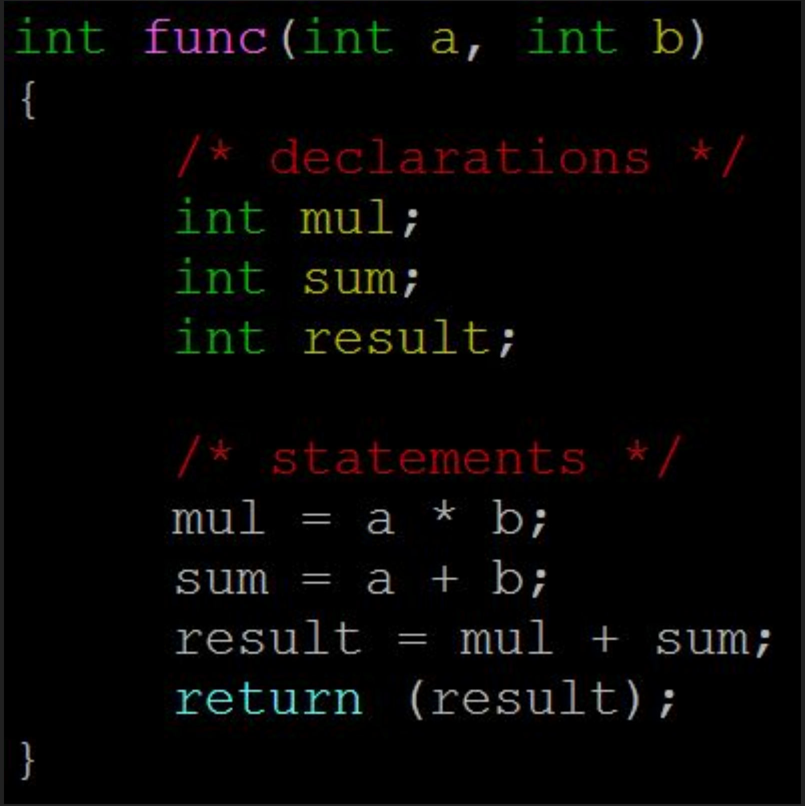
Contains:

- Blocks
- Instructions (assigning values, computing)
- Control structures (conditional statements, loops)

Blocks | Example



```
{
        /* declarations */
        int mul;
        int sum;
        int result;

        /* statements */
        mul = a * b;
        sum = a + b;
        result = mul + sum;
        return (result);
}
```



```
int func(int a, int b)
{
        /* declarations */
        int mul;
        int sum;
        int result;

        /* statements */
        mul = a * b;
        sum = a + b;
        result = mul + sum;
        return (result);
}
```

Instructions, Expressions

#cisfun

Instructions, expressions

Expression, followed by ;

; (semicolon) is a statement terminator (indicates the end of one logical entity)

Expressions always have a value

Types of expressions:

- Basic expressions (i.e. arithmetic operations)
- Affectations (i.e. assigning a value to a variable)
- Comparisons (i.e. checking if a variable is less than a number)
- Logical operators (i.e. checking two things at the same time)
- Binary operators

Basic expressions

Basic expressions: Numbers, variables, arithmetics, function calls

Numbers can be written in decimal (32), octal (032) or hexadecimal (0x32)

The value of a variable is the value it contains

```
32, 032, 0x32, variable
var1 + var2, (var * 32) / 1024, (expression)
```

The value of a function call is its return value (after execution)

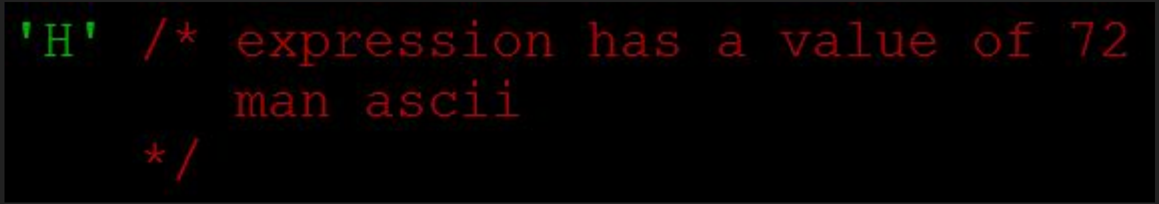
```
func(expression1, expression2 [...]);
func(32, var, var / 32);
```

Characters

‘letter’ (single quotes)

The value of a character is its ASCII code (man ascii)

Example:



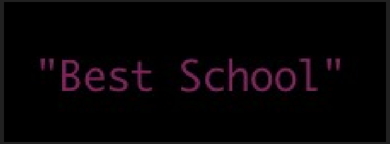
```
'H'  /* expression has a value of 72
        man ascii
     */
```

Strings

“string” (double quotes)

Evaluates to the address in memory of the first character of the string

Example:



```
"Best School"
```

When using strings, the computer creates an array of chars, containing the same number of cells as the length of the string + 1, and then fills the array with the ASCII codes of each letter of the string. In the extra cell (the last one) the computer will store the ASCII code of the special character ‘\0’

Arithmetic operators

+ addition, - subtraction, * multiplication, / division, % modulo (integer remainder)

expression operator expression

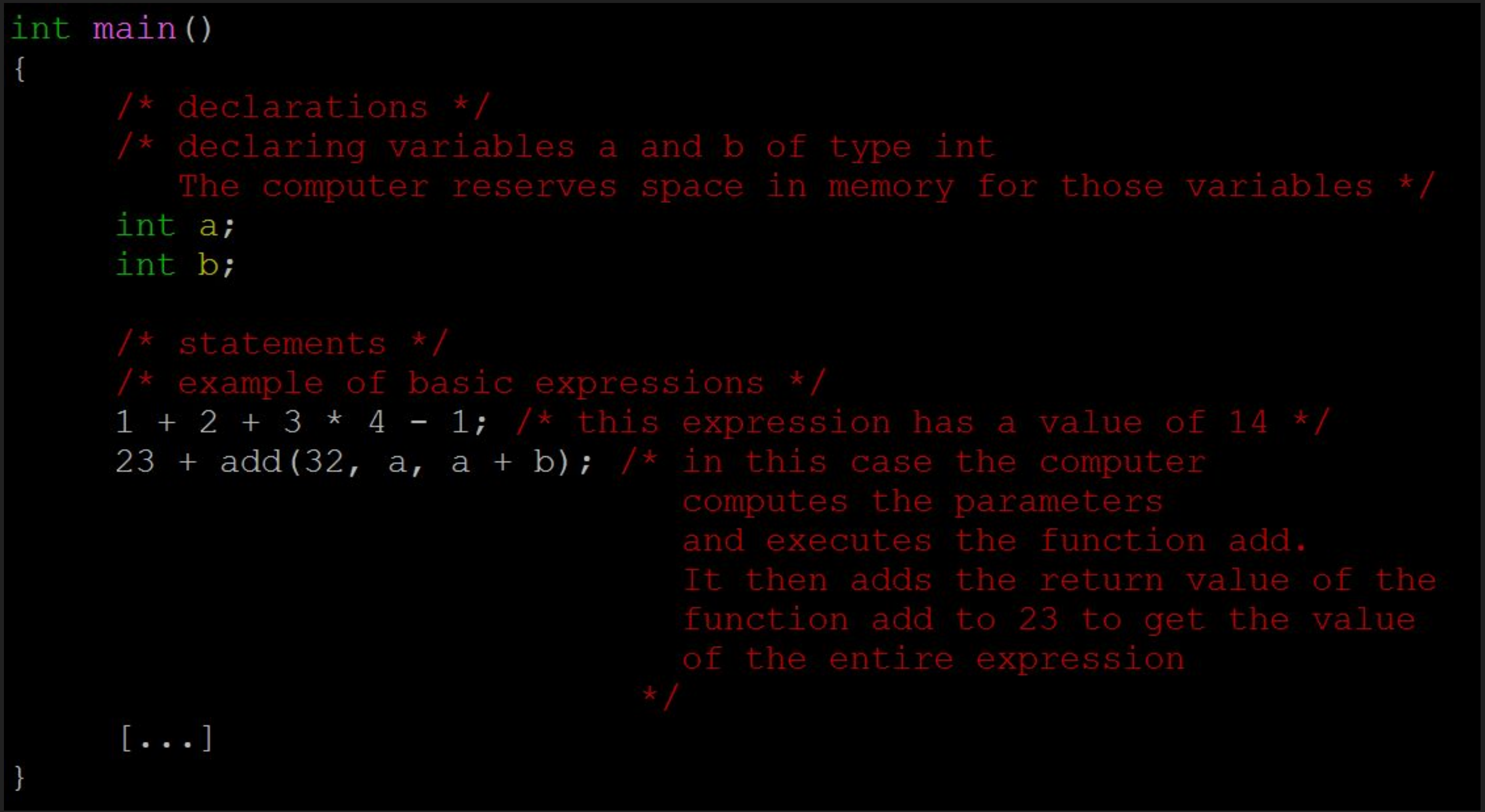
```
1 + 32;
var % 10;
var / var2 + 1337;
```

++ increment, -- decrement

```
/* var is an integer */
++var; /* increments var by 1,
          value of expression is the value of var after the increment */
var++; /* increments var by 1,
          value of expression is the value of var before the increment */
--var; /* decrements var by 1,
          value of expression is the value of var after the decrement */
var--; /* decrements var by 1,
          value of expression is the value of var before the decrement */
```

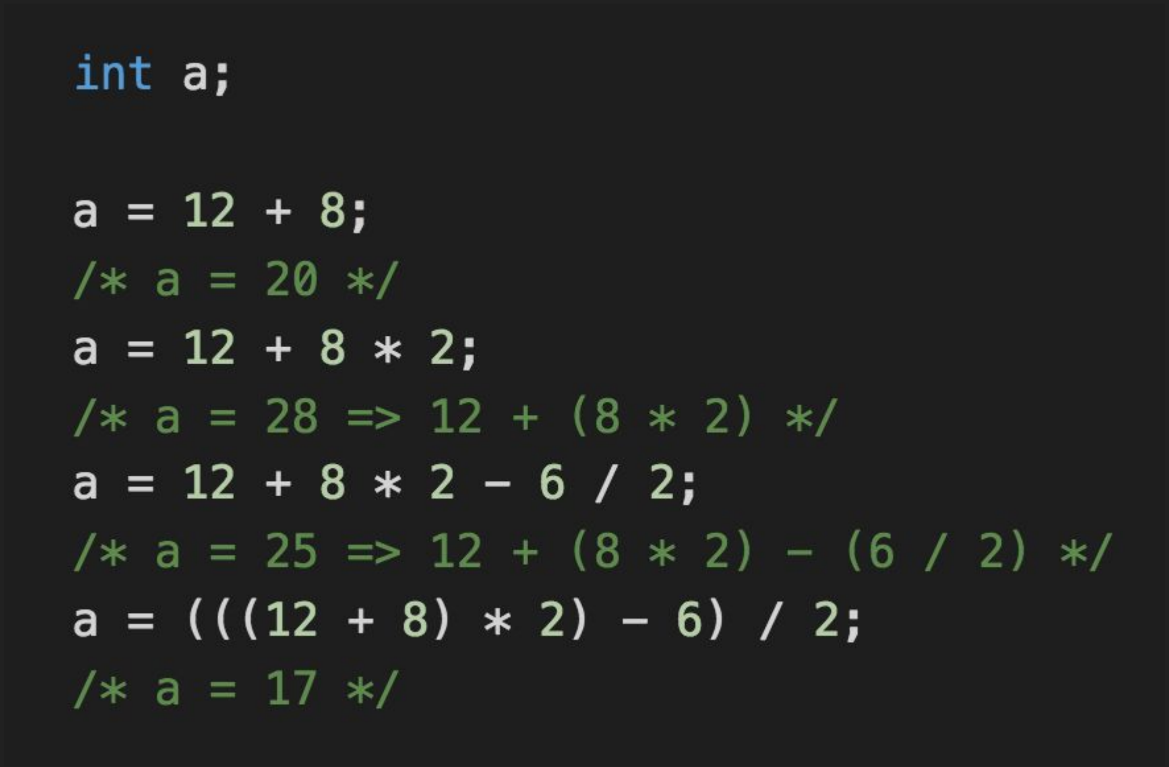
```
int i;
int r;

i = 0;
r = i++; /* r = 0 */
/* i = 1 */
r = ++i; /* i = 2, r = 2 */
```

Basic expressions | example



```
int main()
{
     /* declarations */
     /* declaring variables a and b of type int
        The computer reserves space in memory for those variables */
     int a;
     int b;

     /* statements */
     /* example of basic expressions */
     1 + 2 + 3 * 4 - 1; /* this expression has a value of 14 */
     23 + add(32, a, a + b); /* in this case the computer
                                computes the parameters
                                and executes the function add.
                                It then adds the return value of the
                                function add to 23 to get the value
                                of the entire expression
                              */
     [...]
}
```

Using parentheses with operators



```c
int a;

a = 12 + 8;
/* a = 20 */
a = 12 + 8 * 2;
/* a = 28 => 12 + (8 * 2) */
a = 12 + 8 * 2 - 6 / 2;
/* a = 25 => 12 + (8 * 2) - (6 / 2) */
a = (((12 + 8) * 2) - 6) / 2;
/* a = 17 */
```

Affectations

Change the content of a variable (update the value in memory)

Syntax:
```
var_name = expression;
```

Examples:
```
a = 1;
a = 32 + 3;
b = a + c;
c = func(1024, a + b);
```

If the expression is another affectation, then its value is the value affected to the previous variable

```
a = b = 32; /* the expression b = 32 has a value of 32
               so a = 32 */
/* don't do this at home  */
```

Affectations | elements of arrays and structures

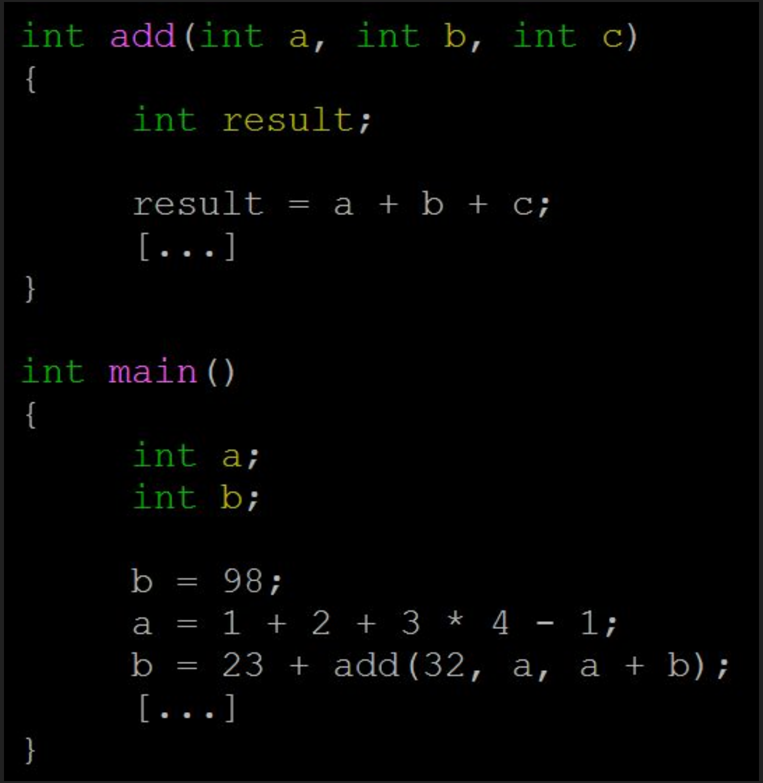
Syntax:

```
var_array[expression] = expression;
var_struct.field_name = expression;
```

Examples:

```
a[0] = b - 32;
a[b + 1] = 1337;
s.age = 32 / 2 + f();

s.p[3] = 0;
aa[98].t[b * 2] = f2(b + 1, 1337 / 2) + 402;
```

Variables assignment | example



```c
int add(int a, int b, int c)
{
      int result;

      result = a + b + c;
      [...]
}

int main()
{
      int a;
      int b;

      b = 98;
      a = 1 + 2 + 3 * 4 - 1;
      b = 23 + add(32, a, a + b);
      [...]
}
```

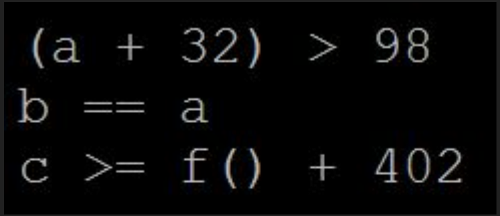
Comparisons

<	Less than
<=	Less or equal to
==	Equal to
>=	Greater or equal to
>	Greater than
!=	Different than

expression operator expression

The value of a comparison is 0 if false, and something else if true.

Examples:



```
(a + 32) > 98
b == a
c >= f() + 402
```

Logical operators

|| OR, && AND, ! NOT

expression1 || expression2 - if one of the 2 expressions is true, then the whole expression is true

expression1 && expression2 - if one of the 2 expressions is false, then the whole expression is false

!expression - if the expression is false, then the whole expression is true. If the expression is true, then the whole expression is false.

The value of the whole expression is 0 if false, something else if true

Logical operators

AND &&

	0	**1**
0	0	0
1	0	1

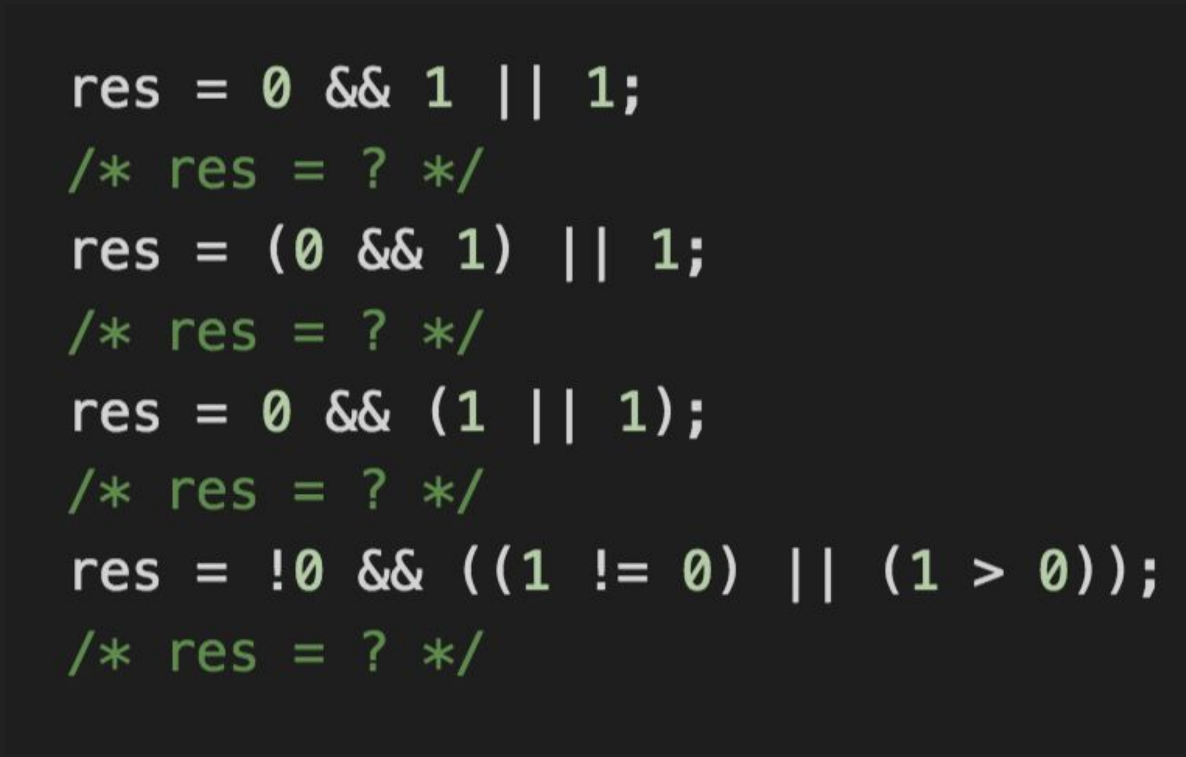
OR ||

	0	**1**
0	0	1
1	1	1

NOT !

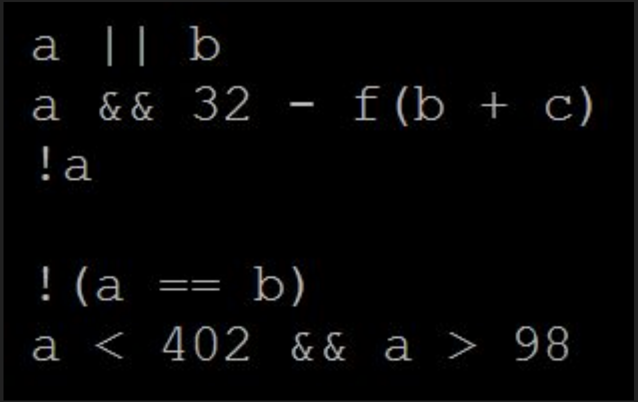
0	1
1	0

Can you solve this?



```
res = 0 && 1 || 1;
/* res = ? */
res = (0 && 1) || 1;
/* res = ? */
res = 0 && (1 || 1);
/* res = ? */
res = !0 && ((1 != 0) || (1 > 0));
/* res = ? */
```

Logical operators | examples



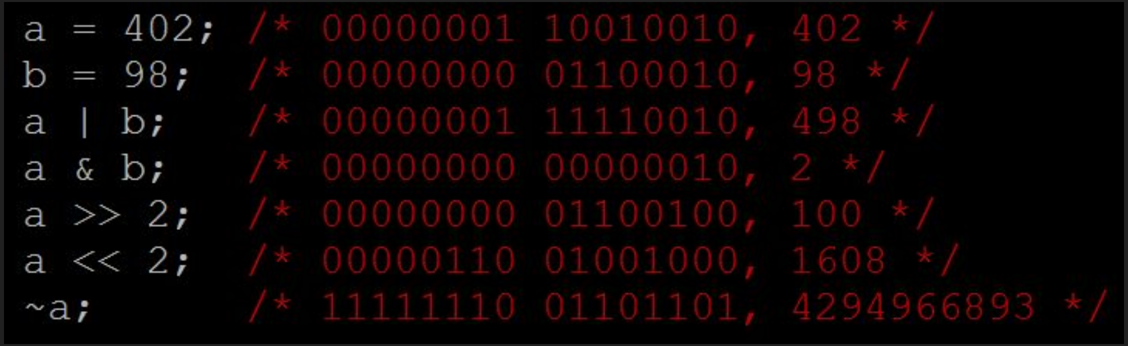
```
a || b
a && 32 - f(b + c)
!a

!(a == b)
a < 402 && a > 98
```

Binary operators

Bitwise operations

| OR, & AND, << LEFT SHIFT, >> RIGHT SHIFT, ^ XOR
~ NOT

expression operator expression
~expression



```
a = 402; /* 00000001 10010010, 402 */
b = 98;  /* 00000000 01100010, 98 */
a | b;   /* 00000001 11110010, 498 */
a & b;   /* 00000000 00000010, 2 */
a >> 2;  /* 00000000 01100100, 100 */
a << 2;  /* 00000110 01001000, 1608 */
~a;      /* 11111110 01101101, 4294966893 */
```

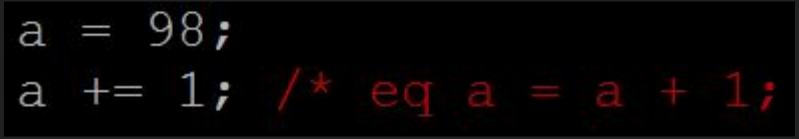
Can you solve this?

```
char b1 = 0b00000010; /* = 2 */
char b2 = 0b00000011; /* = 3 */
char b3;

b3 = b2 & b1;
/* b3 = ? */
b3 = (b2 >> 2) & 0b00000001;
/* b3 = ? */
b3 = (b2 >> 1) & 0b00000001;
/* b3 = ? */
```

More affectations: Compound assignment operators

Perform the operation specified by the additional operator, then assign the result to the left operand

Example:
expression1 += expression2, is equivalent to:
expression1 = expression1 + expression2



```
a = 98;
a += 1; /* eq a = a + 1;
```

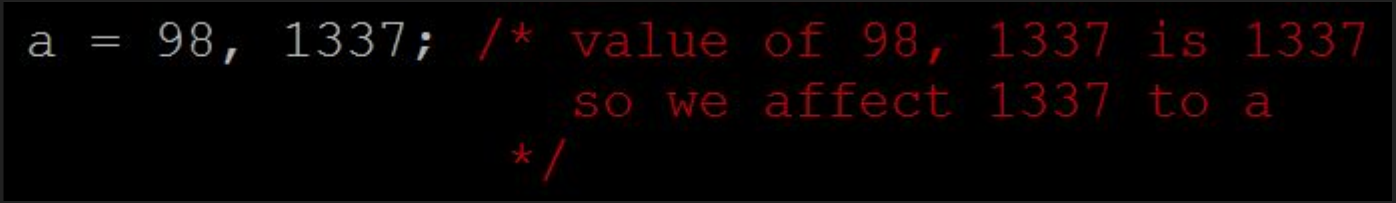
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=

Comma operator

expression1, expression2

The value of the whole expression is the value of expression2

Example:



```
a = 98, 1337; /* value of 98, 1337 is 1337
                 so we affect 1337 to a
                */
```

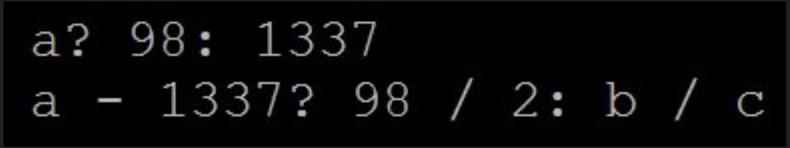
Don’t use this at home

Ternary conditional

expression1? expression2: expression3

Evaluates to expression2 if expression1 is true, and to expression3 otherwise

Example:

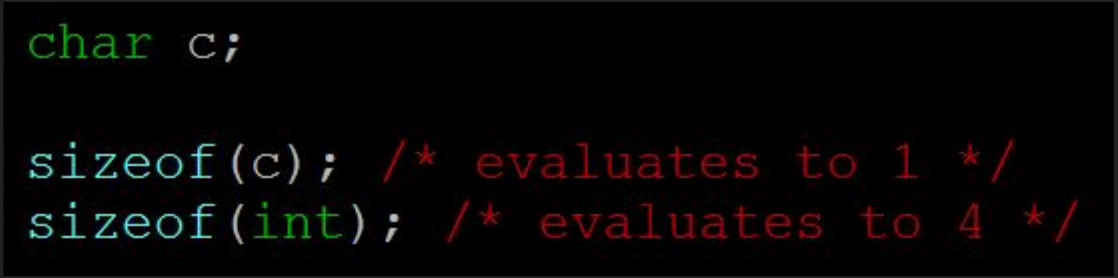


```
a? 98: 1337
a - 1337? 98 / 2: b / c
```

sizeof

Unary operator that evaluates to the size in memory of the variable or type, in bytes

Example:



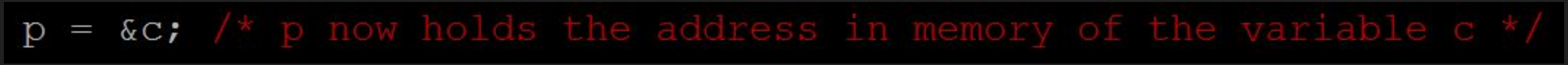
```
char c;

sizeof(c); /* evaluates to 1 */
sizeof(int); /* evaluates to 4 */
```

&

&var_name

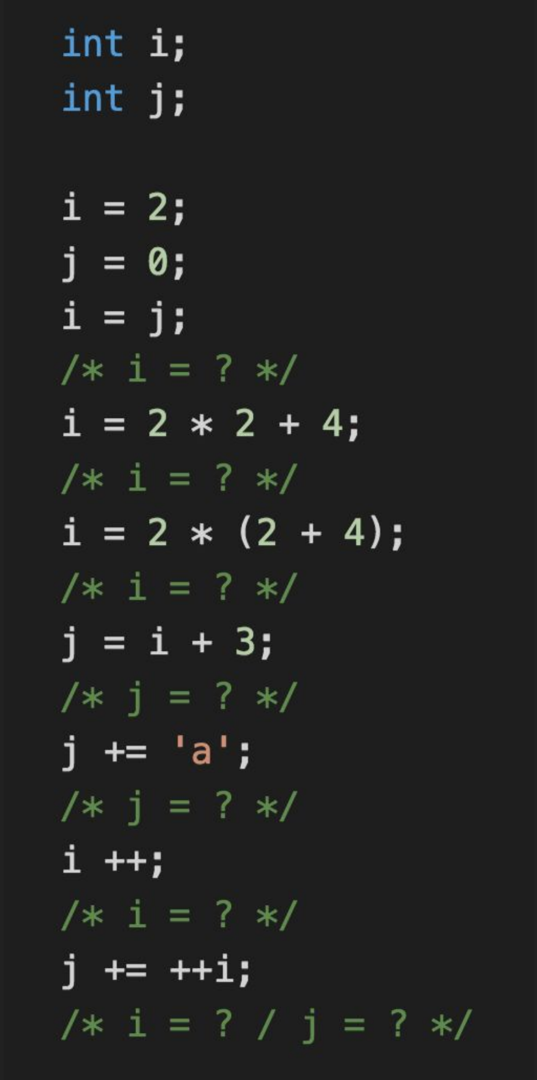
The address in memory of the variable var_name

Example:



```
p = &c; /* p now holds the address in memory of the variable c */
```

Can you solve this?

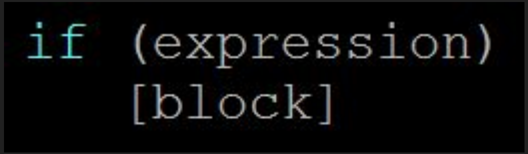


```
int i;
int j;

i = 2;
j = 0;
i = j;
/* i = ? */
i = 2 * 2 + 4;
/* i = ? */
i = 2 * (2 + 4);
/* i = ? */
j = i + 3;
/* j = ? */
j += 'a';
/* j = ? */
i ++;
/* i = ? */
j += ++i;
/* i = ? / j = ? */
```

Control structures

#cisfun

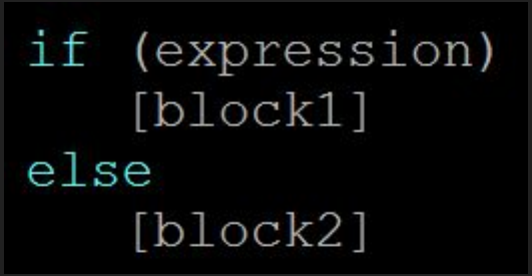
if



```
if (expression)
    [block]
```

If the expression is true (the value of the expression is not 0) then the block is executed

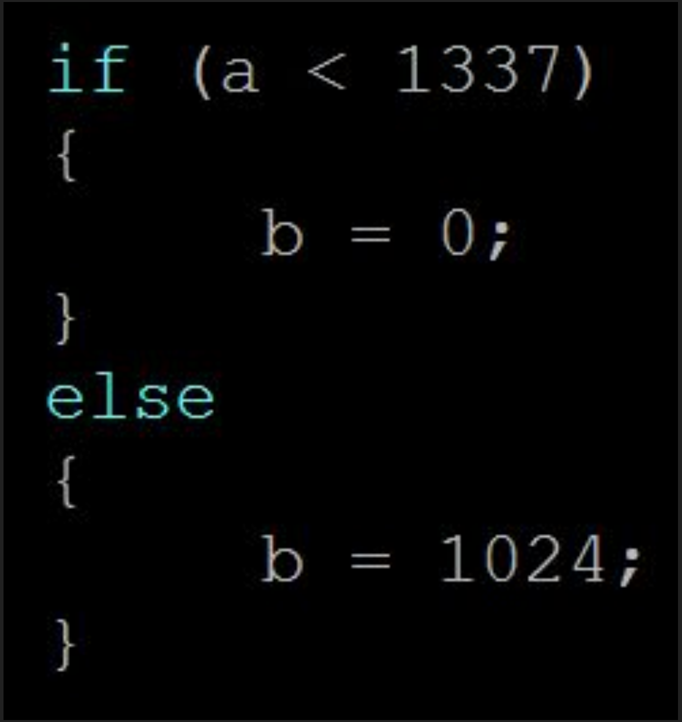
if … else



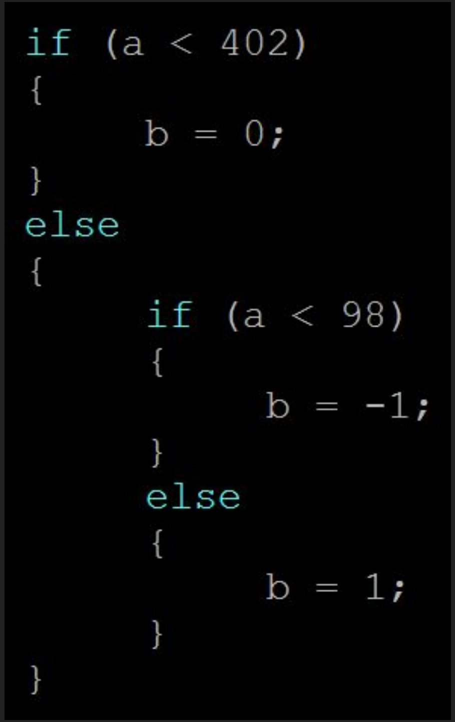
```
if (expression)
    [block1]
else
    [block2]
```

If the expression is true, then block1 is executed. Otherwise, block2 is executed

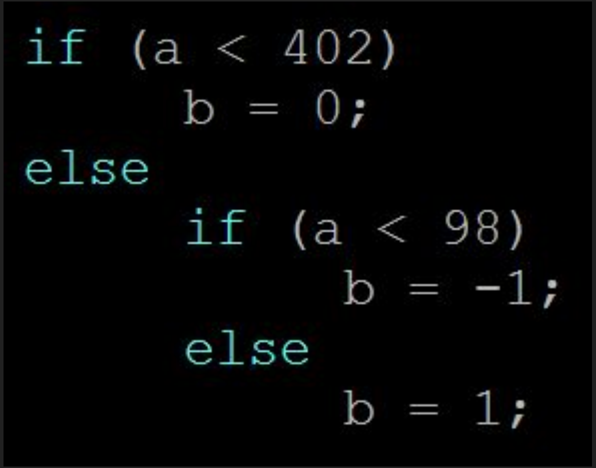
if … else | examples



```
if (a < 1337)
{
        b = 0;
}
else
{
        b = 1024;
}
```

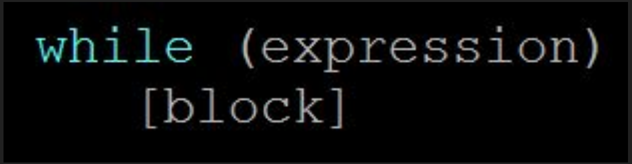


```
if (a < 402)
{
        b = 0;
}
else
{
        if (a < 98)
        {
                b = -1;
        }
        else
        {
                b = 1;
        }
}
```



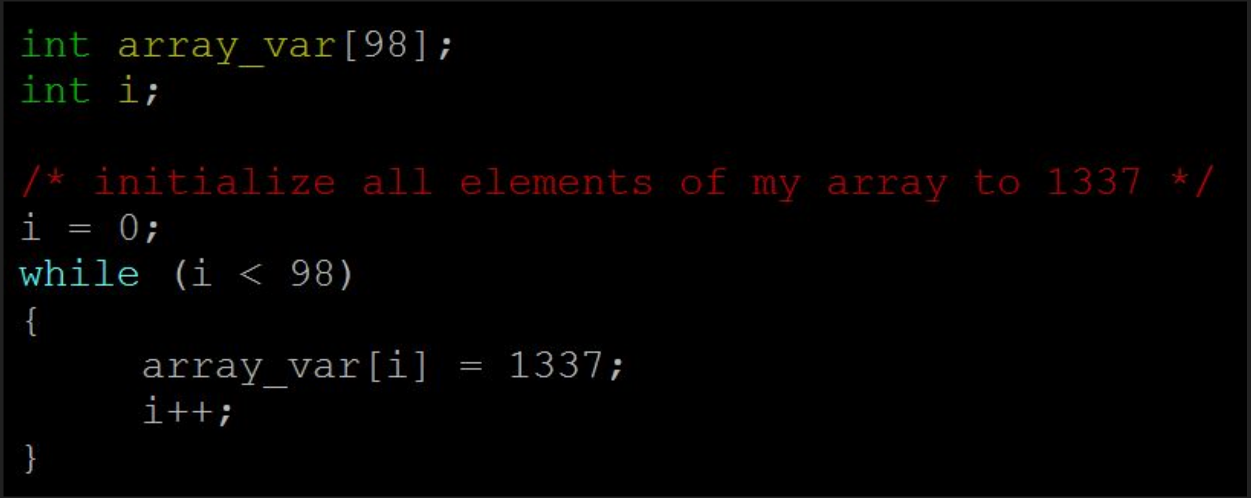
```
if (a < 402)
        b = 0;
else
        if (a < 98)
                b = -1;
        else
                b = 1;
```

while loops



```
while (expression)
    [block]
```

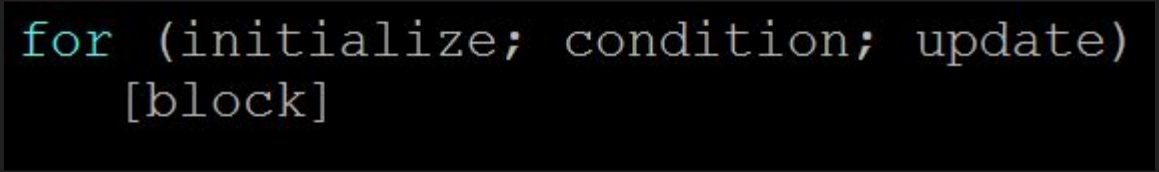
The while loop lets you repeat a block until a specified expression becomes false.



```
int array_var[98];
int i;

/* initialize all elements of my array to 1337 */
i = 0;
while (i < 98)
{
     array_var[i] = 1337;
     i++;
}
```

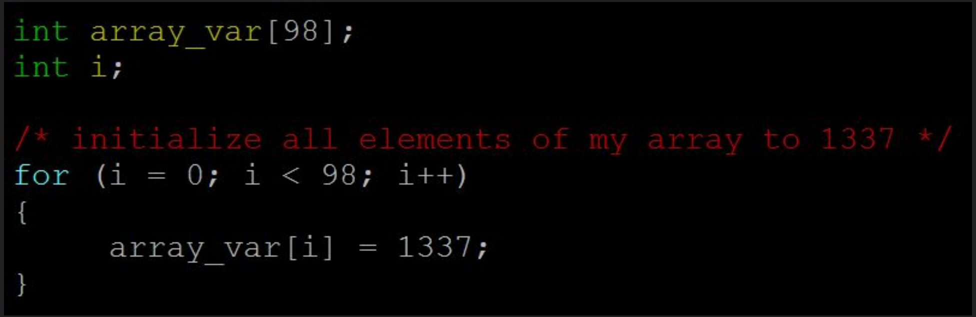
for loops

The for statement lets you repeat a block a specified number of times. The block of a for statement is executed zero or more times until an optional condition becomes false. You can use optional expressions within the for statement to initialize and change values during the for statement's execution.

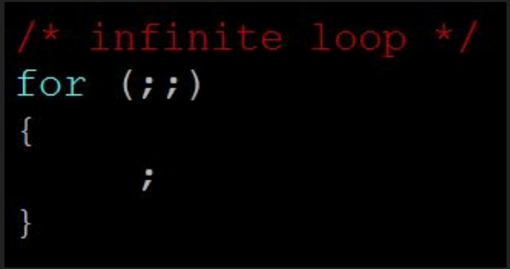


```
for (initialize; condition; update)
    [block]
```

Initialize, conditions and update are optional.

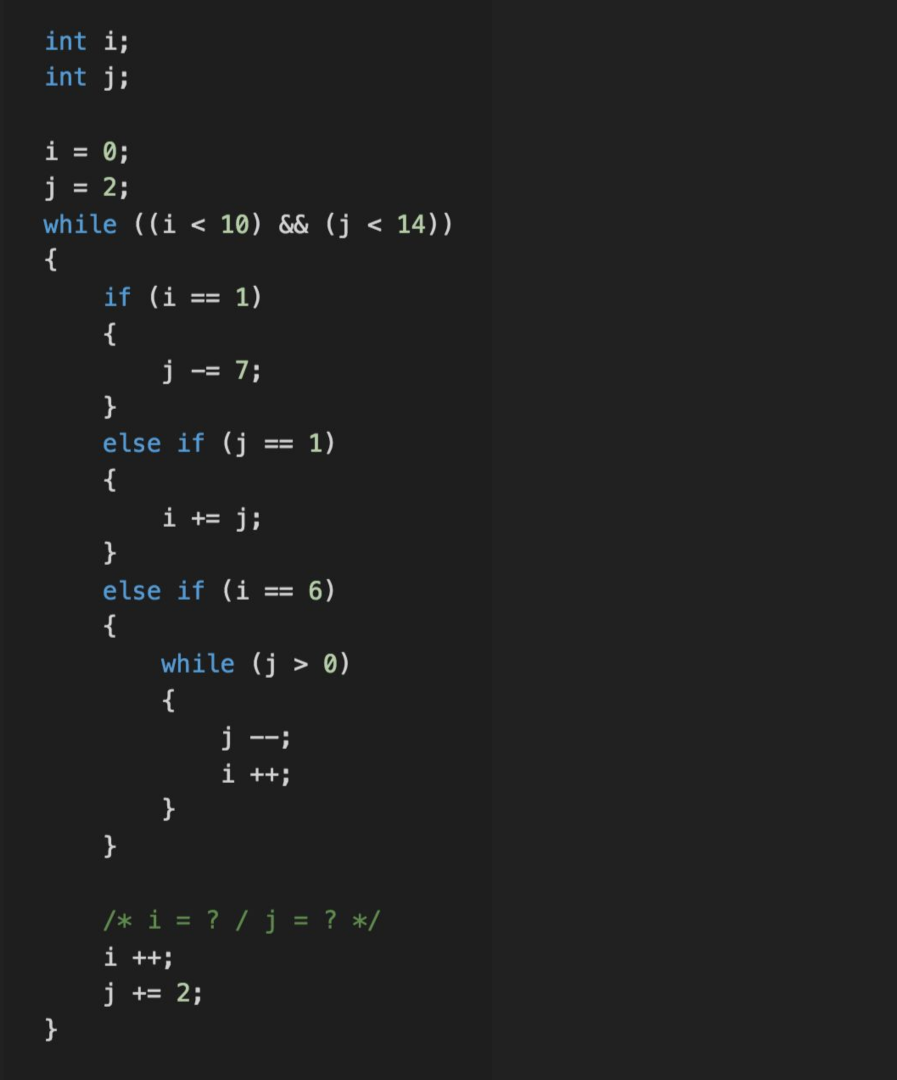


```
int array_var[98];
int i;

/* initialize all elements of my array to 1337 */
for (i = 0; i < 98; i++)
{
     array_var[i] = 1337;
}
```



```
/* infinite loop */
for (;;)
{
     ;
}
```

Can you solve this?



```
int i;
int j;

i = 0;
j = 2;
while ((i < 10) && (j < 14))
{
    if (i == 1)
    {
        j -= 7;
    }
    else if (j == 1)
    {
        i += j;
    }
    else if (i == 6)
    {
        while (j > 0)
        {
            j --;
            i ++;
        }
    }

    /* i = ? / j = ? */
    i ++;
    j += 2;
}
```

return

```
return (expression); /* expression is the return value of the function */
return;
```

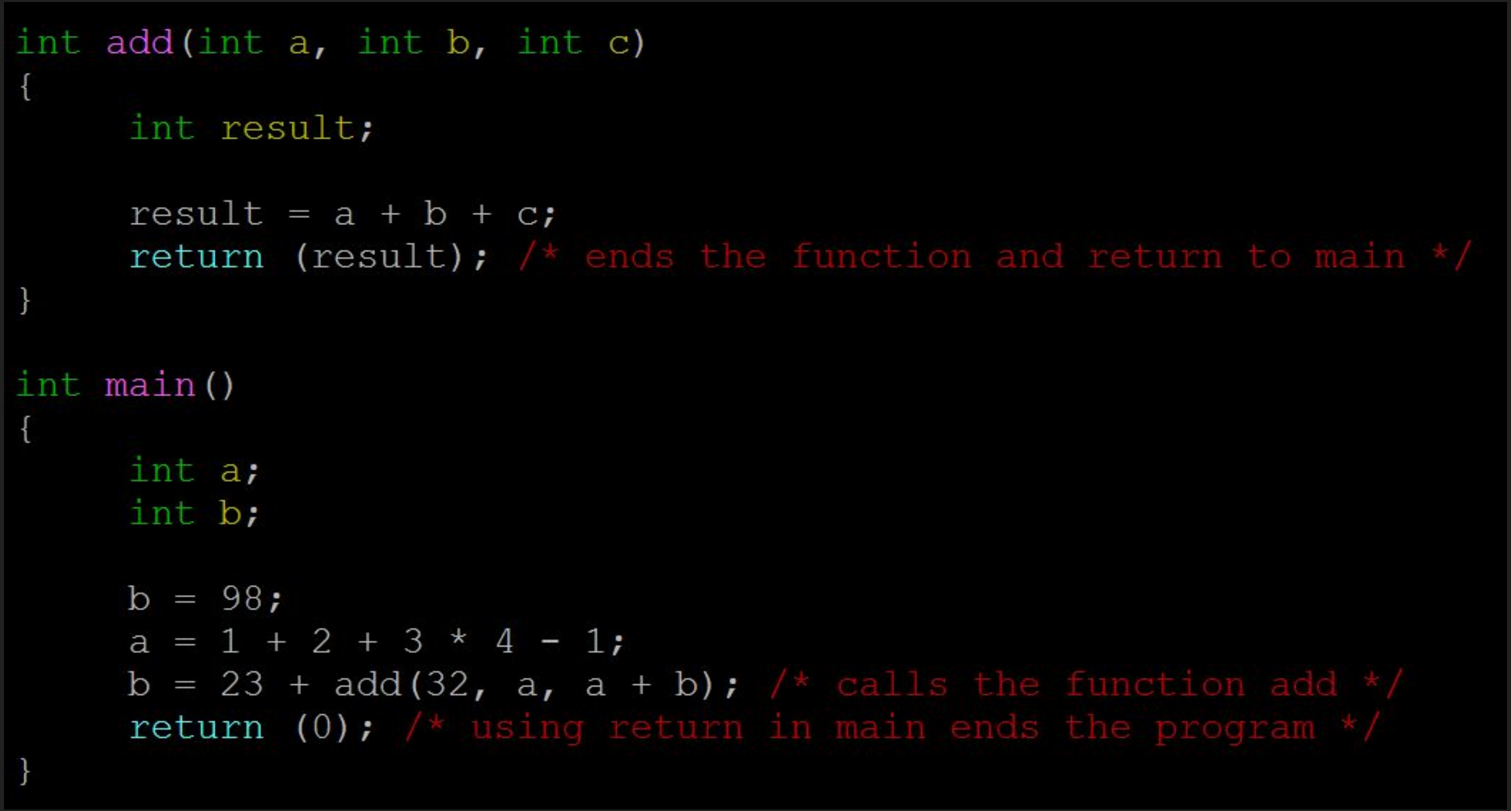
Ends the function and returns to the calling function

If used with an expression, the expression becomes the return value of the function

The type of expression must match the return type of the function

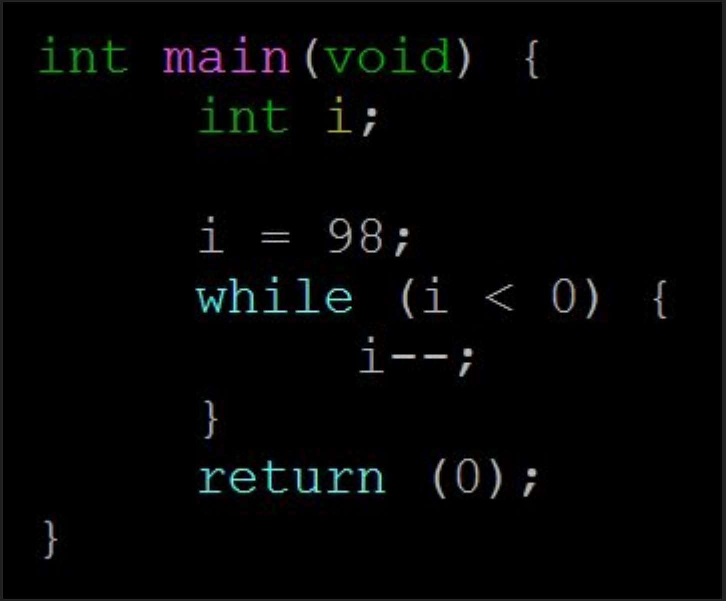
Any code after return will never be executed

return | example

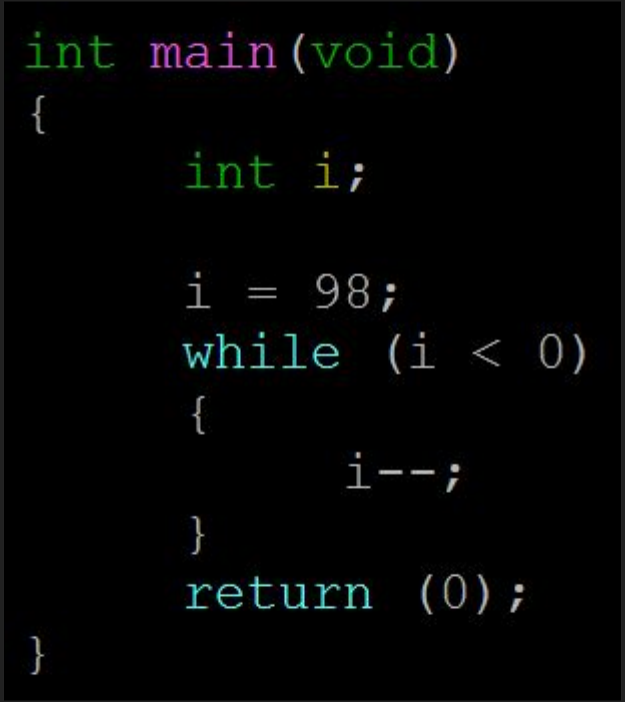


```
int add(int a, int b, int c)
{
      int result;

      result = a + b + c;
      return (result); /* ends the function and return to main */
}

int main()
{
      int a;
      int b;

      b = 98;
      a = 1 + 2 + 3 * 4 - 1;
      b = 23 + add(32, a, a + b); /* calls the function add */
      return (0); /* using return in main ends the program */
}
```

{ } syntax



```
int main(void) {
	int i;

	i = 98;
	while (i < 0) {
		i--;
	}
	return (0);
}
```



```
int main(void)
{
	int i;

	i = 98;
	while (i < 0)
	{
		i--;
	}
	return (0);
}
```

Example

#cisfun

Source code flow of execution

```c
int add(int a, int b)
{
    int sum;

    sum = a + b;
    return (sum);
}

int main(void)
{
    int i;
    int j;
    int res;

    i = 2;
    j = 4;
    res = add(i, j);
    printf("%d\n", res);
    return 0;
}
```

1) Start by the main() function
2) Allocate 4 bytes for i
3) Allocate 4 bytes for j
4) Allocate 4 bytes for res
5) Set the value 2 to i
6) Set the value 4 to j
7) Call the function add():
    a) Allocate frame memory for the function
    b) Copy the value of i and set to a
    c) Copy the value of j and set to b
8) Allocate 4 bytes for sum
9) Compute a + b expression
10) Set the value of the expression a + b to sum
11) Return the value of sum and go back to the main()
12) Destroy the function add() in memory
13) Set the result (return value) of add() to res
14) Call the function printf() (standard library function)
15) return 0 = stop the program

```
return (0);
```

#cisfun